

CloudTree: A Library to Extend Cloud Services for Trees

Yun Tian*, Bojian Xu*, Yanqing Ji†, Jesse Scholer*

*Department of Computer Science, Eastern Washington University, Cheney, WA 99004, USA

† Department of Electrical and Computer Engineering, Gonzaga University, Spokane, WA 99258, USA
{ytian,bojianxu}@ewu.edu, ji@gonzaga.edu, jscholer@ewu.edu

Abstract—In this work, we propose a library that enables on a cloud the creation and management of tree data structures from a cloud client. As a proof of concept, we implement a new cloud service *CloudTree*. With *CloudTree*, users are able to organize big data into tree data structures of their choice that are physically stored in a cloud. We use caching, prefetching, and aggregation techniques in the design and implementation of *CloudTree* to enhance performance. We have implemented the services of Binary Search Trees (BST) and Prefix Trees as current members in *CloudTree* and have benchmarked their performance using the Amazon Cloud. The idea and techniques in the design and implementation of a BST and prefix tree is generic and thus can also be used for other types of trees such as B-tree, and other link-based data structures such as linked lists and graphs. Experimental results show that *CloudTree* is useful and efficient for various big data applications.

Keywords—Cloud Storage; Tree Storage in Cloud; Big Data Structures; External Tree; *CloudTree*.

I. INTRODUCTION

Due to the ever-increasing data size caused by advances in electronic and computational technologies, computer scientists have been making efforts in finding efficient solutions to challenged involving large-scale data. Such efforts in this direction include external memory computing [1], [2], cache-oblivious computing [3], succinct and compressed data structures [4], and distributed and parallel computing [5], [6].

Cloud computing proposed in late 2000s is an idea extended from distributed computing. It centralizes the computing resources and their management into one place (logically) and provides the usage of the resources as “utility services”. By doing so, users are freed from complexities in hardware infrastructure management and only pay for the exact cost of the amount of resources that they need. Since the hardware infrastructure of the cloud is hidden from the user, the data storage capacity and computing power provided by the cloud is conceptually unlimited, which is critically important in the era of big data.

On the other hand, because users do not have control of the low-level hardware infrastructure, it becomes inefficient or impossible for users to perform certain computations. For example, if one wants to create and use an indexing tree data structure for a large data set, the current solution that is available for the user is to rent a physical or virtual machine (VM) from the cloud, then treat the VM as their own local machine and do the computation as needed on the

VM, which conceptually provides unlimited storage capacity and computing power as long as the user continues to pay for the rental. However, there are two critical disadvantages of this solution: 1) The rental of a VM is often more expensive than the mere data storage space rental, provided the same amount of storage space is rented. 2) The user has to pay the rental of the VM, even if no computation work is being conducted, because once the lease of the VM is over, all the data stored within the VM may be permanently lost.

Trees are important linked data structures. When data size is large, often it is necessary to organize the dataset into a tree structure to avoid a linear scan. For example, we may frequently search a set of common prefix strings in a large group of target strings, where a prefix tree is quite useful. In another scenario, in fields of graphics and particle physics, numerous objects with three dimensional coordinates are queried and processed on a regular basis, in which a k-d tree or an R-tree is quite useful to organize these spatial data objects. Again, existing cloud storage services that provide underlying hashing, sorting or search tree indices are not effective for these *spatial* data items. In these cases, users have to build their own trees in a cloud.

In this work, we propose a library that enables a new cloud service *CloudTree*. With *CloudTree*, users are able to create, manage, and use tree data structures for big data sets on a cloud without using VM. Users will perform the computation of their application on their local machine and treat the cloud as their extended memory, where the tree is being stored. We make the following contributions.

- 1) To our best knowledge, there is no existing cloud storage service that directly allow users to create, manage, and use an *external* tree on cloud. We are first to propose this concept and implement it as a cloud client-side library.
- 2) Because of the scalability and unlimited storage space offered by cloud, *CloudTree* enables users to create trees of *unlimited* size in their applications. In this case, users need not be concerned about the limited size of their local RAM.
- 3) We adopt several optimization techniques to address the challenge of expensive network communications between an user’s local computer and a cloud. We utilize caching, prefetching, and aggregated operations, which significantly improve the performance of *CloudTree*. We implement and benchmark the performance of *CloudTree* using the Amazon Cloud. Experimental results suggest *CloudTree* is promising

and can be quite useful in various big data applications, including performance-demanding applications.

4) The idea of CloudTree can be easily used for other link-based data structures, such as linked list and graphs.

II. RELATED WORK

Much research has been focused on utilizing tree-based structures in the cloud for the purposes of indexing. One such work involves the use of an A-tree [7]: a distributed architecture that combines bloom filters and R-trees for fast index queries. The work in [8] also utilizes the cloud to store a distributed tree in an attempt to achieve real-time on-line analytical processing. Similar to CloudTree, the work in [9] uses a B-tree contained in a NoSQL database (MongoDB) for indexing. However, none of these works allow for users to manipulate the data, nor is the tree generic. Each utilize specific tree structures whose main purpose is for indexing, not for general use.

A number of related works allow for the creation of tree structures on Amazon's EC2 service. The solution proposed in [10] is that of a dynamic B-tree indexing scheme that resides in the cloud. HSQL [11] similarly utilizes NoSQL databases to store a distributed B-tree. In [12], a tree-based structure is proposed that ensures database delete operations are external and permanent. One major drawback to these works is their reliance upon Amazon EC2; because the structure is implemented on a virtual machine in main memory, it is not accessible from outside the cloud. CloudTree allows for external storage of any generic tree structure that is permanent without the need for a constantly running virtual machine.

Rather than focusing on cloud tree storage, much research is devoted to indexing large amounts of data in a tree structure stored on external memory. B and B+ trees [13] have had a long history of success in indexing large data sets on external memory. String B-tree [14] and its cache-oblivious version [15] serve the indexing of large strings for pattern matching on external memory. Behm et al. [16] proposes a combination of tree and index list to quickly query indexes stored in external memory to balance I/O and query time. This related research attempts to confront similar challenges in storing tree structures on external memory that allow for fast creation, querying, and updating. However, these solutions lack the scalability that CloudTree provides, as well as integration with cloud services.

III. DESIGN OF A CLOUDTREE

In this section, we describe general concepts about CloudTree, including API design, CloudTree node representation, generic design and advantages of CloudTree.

A. API Design

Figure 1 shows a piece of sample code that calls the CloudTree APIs. At lines 4 and 5, we create a cloud

```

1 import ewu.ytian.CloudTree;
2 .....
3 // Note CloudTree is the parent class of CloudPrefixTree and CloudBSTree
4 CloudTree ctrie = new CloudPrefixTree(treeName1, newTreeOrNot);
5 CloudTree cbst = new CloudBSTree(treeName2, newTreeOrNot);
6 ctrie.init();
7 cbst.init();
8 ctrie.insert('phrase1');
9 cbst.insert(num1);
10 .....
11 ctrie.query('phrase1'); //return true or false
12 cbst.delete(num3);
13 .....
14 ctrie.delete('phrase3'); //delete phrase3 from the tree ctrie.
15 cbst.query(num1); //return true if num1 exists in the tree cbst.
16 .....
17 ctrie.close(); //paired with init() call
18 cbst.close(); //paired with init() call

```

Figure 1. Sample code using CloudTree APIs to create and manipulate trees in a cloud.

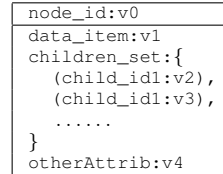


Figure 2. The design of a CloudTree node, where each tree node is identified by a numerical tree node id.

prefix tree instance *ctrie* and a cloud binary search tree instance *cbst* respectively. We provide a string *treeName_i* to uniquely identify each tree in cloud. We can also specify whether the tree instance will be initialized using an existing tree in cloud by providing a second parameter *newTreeOrNot*.

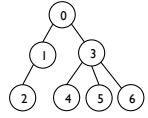
In figure 1, the *init()* method performs authentication and sets up parameters used to communicate with the cloud. Then, users can use the CloudTree instance as if it is stored in local RAM, to insert, query and delete data items in the tree. After finishing all operations, the user is required to call the *close()* method to clean up resources and to permanently save the tree name and other information.

B. CloudTree Node Representation

Each CloudTree node is modeled as an object, similar to a JSON object. Figure 2 shows the general design for a CloudTree node. We use a collection of attribute-value pairs to describe a tree node. For example, in the top box of figure 2, we identify each node with a pair (*node_id* : *v₀*), where an attribute named *node_id* — together with a numerical value *v₀* — are used as a reference to uniquely determine a tree node. In addition, if needed, each node has an attribute *data_item* to describe a data item (or a key) stored in a node.

Another important attribute in a CloudTree node is *children_set* which records a set of child references and, if needed, the data item stored in each children. The child set is shown in figure 2 in a format $\{(child_id_1 : v_1), (child_id_2 : v_2), \dots\}$, where *child_id_i* refers to a numerical child tree node id, and *v_i* denotes the data item stored in that child.

It is quite flexible to add other attributes in a CloudTree node as needed. For example, an attribute *leaf* describes whether the node is a leaf node or not. In another example, for a prefix tree, we need an attribute *word* to indicate



(a) A generic tree structure.

Tree Node ID	Other attributes and values in each node.
0	children_set={1, 3}; data_item={11, 56, 89}
1	children_set={2}; data_item=...
2	children_set={}; data_item=...; leaf=true
3	children_set={4, 5, 6}; data_item=...
4	children_set={}; data_item=...; leaf=true
5	children_set={}; data_item=...; leaf=true
6	children_set={}; data_item=...; leaf=true

(b) CloudTree representation.

Figure 3. A generic tree is shown in figure(a). Figure(b) shows its CloudTree representation.

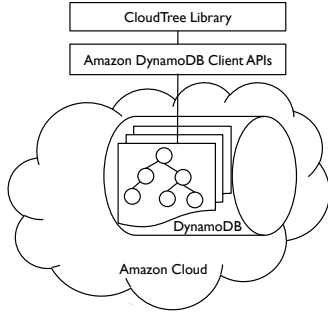


Figure 4. Overview of the CloudTree Library Implementation. A CloudTree instance is stored in an Amazon DynamoDB database table, and is manipulated by using Amazon DynamoDB client APIs.

whether the path from the root to the current node constitutes a valid English word in a dictionary.

C. Generic Design

In figure 3, we describe a simple tree structure and its representation using CloudTree. The number in each node in figure (a) is a node id. Traditional child references (or child links) in each node are represented by a set of child node ids. For example, node 3 has three children with ids {4, 5, 6}. Therefore, we can store a CloudTree instance either in a file or in a database table in a cloud, with each tree node considered as an item in the file or in the table. Then, we use the unique tree node id as a primary key to retrieve a tree node. It is worth mentioning that the design can be applied to other linked structures as well, for example, linked lists or graphs.

D. Advantages of CloudTree

- **Generic:** First, the design of the proposed CloudTree is generic, allowing it to be applied to any existing tree data structure, such as prefix tree, B-tree, R-tree, K-d tree, etc. Second, the CloudTree library can be implemented using cloud services from various cloud providers, such as Amazon, RackSpace or other providers.
- **External and Permanent:** A CloudTree instance is not stored in the traditional *main memory (RAM)*, but in

durable cloud storage instead. Note that CloudTree instances are not stored in the RAM of a VM that is provisioned in a cloud either, such as an EC2 instance of Amazon.

- **Unlimited Size:** Due to scalable cloud storage space, CloudTree's use of the underlying cloud storage results in a conceptually unlimited size.
- **As A Cloud Service:** With the proposed CloudTree library, authorized users can connect to a cloud and create a tree data structure in the cloud, anywhere that has Internet access.
- **Dynamic:** CloudTree supports tree creation, query, update and deletion operations. After a CloudTree instance is closed, the user may *re-open* the existing tree again.
- **Transparent:** Users manipulate a CloudTree instance in the same way as if the tree is stored in the RAM of a local computer.

IV. IMPLEMENTATION

In this section, we describe an overview of implementation, the underlying storage and two members of CloudTree: CloudPrefixTree and CloudBSTree.

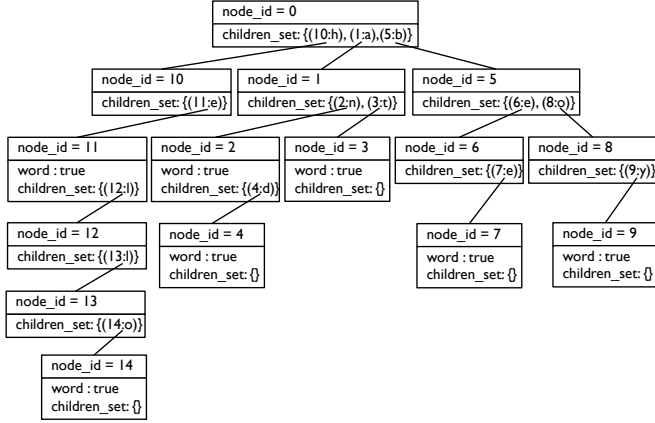
A. Overview

We use Amazon DynamoDB service to implement CloudTree. Figure 4 illustrates the implementation of CloudTree. The CloudTree library extends the underlying Amazon DynamoDB client APIs to enable tree creation, insertion, query and deletion operations in the cloud. Each tree instance is stored in a separate Amazon DynamoDB table.

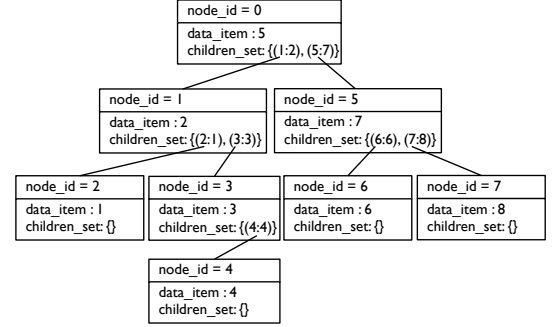
There are three reasons why we chose Amazon DynamoDB among other services. First, access to data in Amazon DynamoDB is fast with low read and write latency, which is quite suitable for performance-demanding applications, such as gaming and shopping cart services [17]. Second, DynamoDB storage is scalable and durable with high availability, two very desirable features that are required by the CloudTree. Third, our CloudTree design nicely fits in the data model that DynamoDB provides, which is discussed in the next section.

B. Data Model of DynamoDB

Amazon DynamoDB is a NoSQL database. Except for the required primary key, a table in DynamoDB does not have a fixed schema. An item or object in such a table could have arbitrary number of attributes, with each attribute represented by a name-value pair. Note that the value associated with an attribute could be a single value or a multi-valued set. These features in DynamoDB provide the means to represent an object that is traditionally stored in the main memory. In this work, we consider each CloudTree node as an item in an DynamoDB table. The DynamoDB provides client APIs to insert, update or delete an item in a DynamoDB table.



(a) An example prefix tree represented using CloudTree, called CloudPrefixTree, after a set of English phrases have been inserted in an order {"an", "at", "and", "bee", "boy", "hello", "he"}.



(b) An example binary search tree represented using CloudTree, called CloudBSTree, after a set of integers have been inserted in an order {5, 2, 1, 3, 4, 7, 6, 8}.

Figure 5. Two instances of CloudTree: a CloudPrefixTree and a CloudBSTree. node_id always increases when new elements are inserted into a CloudTree.

C. Cloud Prefix Tree Implementation

Figure 5(a) presents a CloudPrefixTree instance, a prefix tree represented with the CloudTree design. The tree node id is shown in the top box of each node, on which a hash index has been constructed for fast data retrieval. With provided DynamoDB client APIs, we can retrieve a CloudTree node, given its tree node id.

We explicitly store child node ids for an node using the attribute *children_set*, a set data structure with each element represented by a string. Each element in the *children_set* contains two pieces of information, a child node id and the character that is associated with that child. The two pieces of information share a single string, but is delimited by the special character ':' (colon). For example, there are three children in the root node (*node_id* = 0) in figure 5(a), with child node ids {10, 1, 5}. The corresponding characters associated with these children are {'h', 'a', 'b'}. Note that since we use a string set in DynamoDB for the attribute *children_set*, string elements in the set maintain no order.

We describe the CloudPrefixTree insertion operation in Algorithm 1. The variable *nextNodeID* denotes the next available number for a new tree node id, which is increased by one after each tree node insertion. Pseudo code lines 1 and 2 handle the edge case of root node insertion. If nodes already exist in the tree, we start at the root node with tree node *id* = 0 at line 3. In line 4 through 11, we search whether a prefix string already exists in the cloud tree that matches any prefixes $\{S[0], S[0,1], \dots, S[0,1, \dots, i]\}$ of string *S*. The for loop breaks once the prefix with the maximum length has been found. At line 5, the function *hasChild()* searches in the current tree node whether a child that is associated with S_i exists. It returns a positive *childNodeID* if such a child exists, otherwise it returns a negative number. The

function *addChild(curNodeID, child)* at line 10 adds a child specified by a string parameter *child* into the tree node with *id* = *curNodeID*. We then create new nodes and insert the remaining unscanned characters $S[i+1, i+2, \dots, S.length-1]$ in lines 12 through 15. Lines 16 and 17 insert a leaf node that indicates the end of the string in the tree.

We also implemented the query and deletion operations for the CloudPrefixTree. Due to the limited space, we did not provide its pseudo code here.

D. Cloud Binary Search Tree Implementation

In figure 5(b), we present a binary search tree that is maintained in cloud, named as *CloudBSTree*. The tree is created after inserting a collection of keys of integers.

We use the *node_id* and *children_set* in the same way as the CloudPrefixTree node. For example, the tree node with *node_id* = 0 has two children represented with the *children_set* {"1:2"}, {"5:7"}, with child ids {1, 5} and the data items stored in these children are {2, 7} respectively. Note that an empty *children_set* value means a leaf node.

Unlike a CloudPrefixTree, we add a new attribute *data_item* for each tree node, to describe the data item stored in that tree node. The *data_item* attribute along with the string set *children_set* together to distinguish the left child and the right child within a node, since the *children_set* maintains no order itself. Also, this design allows to retrieve all children's information within a tree node in one cloud access, reducing the number of network communications.

Algorithm 2 describes the insertion operation of a CloudBSTree. Starting with the root node, the while loop at lines 5 to 14 locates the place where the new key will be inserted into the tree. The function *followChild(curNode, n)* at line 10 returns the left child node id of the current node if $n < curVal$, or returns the right child node id if

Algorithm 1: *insert(String S):* insert a prefix string *S* into a CloudPrefixTree.

Input: The string to insert *S*.

```

/* Start with root node, with id = 0. */
1 if nextNodeID = 0 then // root node not exist
2   Create the root node with id =
   0 and insert into cloud DynamoDB table;
3 curNodeID ← 0;
4 for i = 0, 1, ..., S.length - 1 do
   /* Whether in current node there is a
   child associated with Si */
5   childNodeID ← hasChild(curNodeID, Si);
6   if childNodeID ≥ 0 then // Such a child for
   Si exists.
7     curNodeID ← childNodeID;
8   else
9     /* Create a child string. Child id
9     and the character Si associated
10    with the child are delimited with
10    :, then add it to current node. */
10    child ← "nextNodeID : Si";
10    addChild(curNodeID, child) /* Update tree
10    node in cloud via internet. */
11    break;
/* For each character that has not yet
scanned in S */
12 for j = i + 1, i + 2, ..., S.length - 1 do
13   Create a tree node N for Sj, with id =
   nextNodeID and with child id = nextNodeID + 1;
14   Insert N into the cloud DynamoDB table;
15   nextNodeID ← nextNodeID + 1;
16 if j == S.length then // If S is not a prefix
of an existing string in the tree.
17   Insert a leaf node indicating the end of S;

```

Algorithm 2: *insert(n):* Insert a key *n* in the CloudBSTree.

Input: The number *n* to be inserted.

```

/* Start with root node, with id = 0. */
1 curNodeID ← 0;
2 if The first key to insert then
3   Create the root node as a leaf and store it into cloud.
4 else
5   while true do
6     curNode ←
6     get tree node in cloud with curNodeID;
7     curVal ← get the data item in curNode;
8     if curVal == n then // Key already
8     exists in the tree.
9       return;
9     idToGo ← followChild(curNode, n);
10    if idToGo < 0 then // idToGo is not
10    valid, find the place to insert.
11      break;
12    else
13      curNodeID ← idToGo;
14    /* Create a child string. Child id and
14    child data item are delimited with
14    :, then add it to current node. */
15    child ← "nextNodeID : n";
15    addChild(curNodeID, child) /* Update tree
15    node in cloud via internet. */
16    Create a new tree node for key n with id =
16    nextNodeID;
17    Add the new node into cloud DynamoDB table;
18    nextNodeID ← nextNodeID + 1;

```

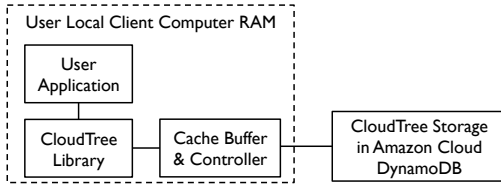


Figure 6. CloudTree with a cache implementation in the RAM of user's client computer.

$n > curVal$, or returns a negative number if *curNode* is a leaf node. Once the place for insertion is located, at lines 15 and 16 we create and add a child string into the current node, which points to the new node for the key *n* that will be inserted at lines 17 and 18. Last, we increment the variable *nextNodeID* for the next insertion.

We also implemented query and deletion operations for the CloudBSTree. Due to the limited space, we did not provide its pseudo code here.

V. OPTIMIZATIONS

We employ several optimization techniques to improve performance of the proposed CloudTree. In particular, caching, prefetching, and aggregation are all effective in dramatically reducing the number of communications via Internet.

A. Caching and Prefetching

The *latency* cost we have to pay for each data access over the Internet could be very high, up to a hundred milliseconds. Therefore, when manipulating a CloudTree instance, it is crucial to reduce the number of data accesses via Internet in order to achieve good performance. Motivated by *caching* techniques for reducing disk I/O operations, we add a cache module to the CloudTree.

Figure 6 illustrates the design of CloudTree with a cache — a bounded cache buffer located in the RAM of a user's client computer and a cache controller. The cache is bridged right between the CloudTree library and CloudTree instances stored in a cloud.

The CloudTree cache here works in a similar way as a cache memory for a traditional hard drive. When user

applications access a tree node in a CloudTree instance, the request is first sent to the cache controller in figure 6. Then the cache controller checks whether the tree node has been cached in the local cache buffer. If so, the copy of the tree node is retrieved from the cache. Otherwise, the cache controller retrieves from the cloud via Internet the requested tree node on user’s behalf, then returns the tree node to the user, as well as saves it into the cache buffer. When updating a tree node in a CloudTree, the cache controller updates the copy of the data first in the cache buffer if available, then updates the data item in cloud as well.

When the cache retrieves data from the cloud, we use *prefetching*. Particularly, when an application reads an uncached tree node with $node_id = x$ — in addition to the requested tree node x — the cache controller prefetches a collection of tree nodes with node ids of $\{x + 1, x + 2, \dots, x + prefetchSize\}$ and stores them into the cache buffer. In this way, it is quite likely that subsequent tree node access requests can be answered with the data in the cache, without communicating to the cloud via Internet for each access request. We adopt the *Least Recently Used (LRU)* policy [18] to discard cached tree nodes when cache is full.

B. Aggregations

Each CloudTree operation (query, insert or delete etc.) may consist of multiple tree node insertions and updates. For example, when inserting a string into a CloudPrefixTree, in a worst case scenario we have to create a tree node for each character in the string and insert into a underlying cloud database table. In this case, we have to pay the Internet latency for *each* tree node insertion, which is not efficient.

We aggregate these small operations or requests into one big request, then send the big request to the cloud. In this way, we only pay the latency cost once for a group of data access or updates. In particular, When the cache controller performs prefetching, it fetches a collection of tree nodes in one transaction by using underlying bulk cloud query operations. Similarly, when inserting or deleting a key from a cloud tree, we aggregate multiple tree node insertions (or deletions) into one bulk insertion (or deletion) operation, thus reducing latency cost. Note that we only aggregate tree node operations within the scope of each *individual* CloudTree operation. In other words, the set of tree node insertions for $tree.insert(keyA)$ and $tree.insert(keyB)$ would NOT be aggregated into one transaction. The usefulness of these optimizations are evaluated in the next section.

VI. EXPERIMENTS

We perform experiments with CloudTree instances stored in Amazon cloud. We verify the usefulness of the CloudTree library and measure time cost of insertion, query, and deletion for two members of CloudTree, the CloudPrefixTree and the CloudBSTree, implemented with Java and the Amazon AWS Java SDK.

CPU	RAM	OS Type	OS Kernel	Java SDK	Amazon AWS SDK	Internet Latency	Internet Bandwidth (upload)	Internet Bandwidth (download)
Intel Core i5 2.5GHz	8GB 1600 MHz	OS X 10.8.5	Darwin 12.6.0	1.8.0_25	1.9.23	74 Millisec	0.65 Mbits/sec	5.88 Mbits/sec

Table I
CONFIGURATION PARAMETERS ON THE CLIENT COMPUTER.

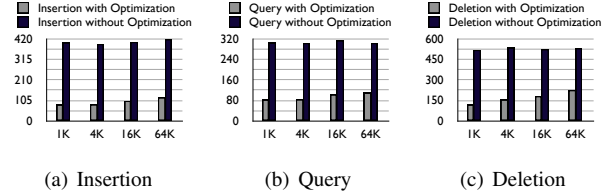


Figure 7. CloudPrefixTree performance using strings of length eight. Horizontal axis denotes various number of strings maintained in the tree. Time cost shown on vertical axis is in milliseconds.

A. Experiment Setup

We initialize and manipulate a CloudTree instance on a client computer, whose configurations are shown in table I. We measure the last three parameters in table I with an cloud speed test service on CloudHarmony.com. *Internet Latency* shows an average round trip time (RTT) between the client computer and an Amazon EC2 instance that is located in the same zone as the Amazon DynamoDB service that we use in the tests. We also obtain bandwidth parameters between the Amazon cloud service and the client computer.

On the side of the cloud provider, we have to set up three parameters for each DynamoDB table. The throughput parameters specify the reserved throughput capacity for a created table [19], a positive number ranging from one to millions. Without specific notice, we use 500 for both read and write capacity. The last parameter *strongConsistency* describes whether we enforce retrieving the most recent value for a data item. To maintain correctness and consistency of the data in a tree, we always use the *strongConsistency = true* option for all data retrieval.

With regard to test data, we download a 200MB text file of English literature from Pizza&Chili¹. We divide the file into 8-character, 16-character or 32-character strings, and use these strings for CloudPrefixTree tests.

For CloudBSTree, we generate datasets in two ways. In the first approach, we create an array A filled with contiguous integers ranging in $[0, H]$. Then we re-order these numbers in A into a list L so that it results in a *balanced* binary search tree after we insert items in L in order. In the second approach, after A is created, we *randomly* shuffle A to generate an array B . We insert elements in B into CloudBSTree instances.

¹<http://pizzachili.dcc.uchile.cl/texts.html>

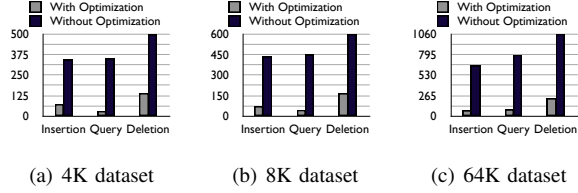


Figure 8. CloudBSTree performance using datasets that result in balanced binary trees. Time cost shown on vertical axis is in milliseconds.

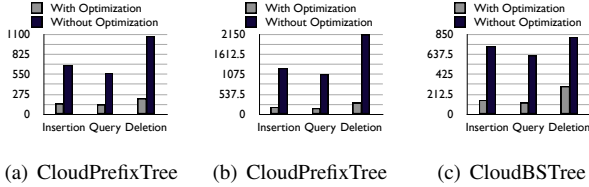


Figure 9. Subfigure (a) and (b) show CloudPrefixTree performance using string length of 16 and 32 respectively. The total number of strings in tests is 16 thousand. Subfigure (c) illustrates CloudBSTree tests with a *random* dataset of 32 thousand unique integers. Time cost shown on vertical axis is in milliseconds.

When measuring performance, insertion time cost is an average time cost of the insertion for an entire dataset. Query and deletion time cost is measured as an average using 200 *random* data items in the tree. When performing tests with optimization enabled, if without special notice, we use a value 25 for the parameter *prefetchSize* and a cache buffer with size of 10000 cache lines. Each cache line is used to cache one tree node.

B. CloudPrefixTree Performance

In figure 7, we show the performance of CloudPrefixTree. We perform tests using various number of strings, with string length 8. It takes roughly 100 milliseconds to insert and query a string of length 8 in the tree when optimizations are enabled, while it takes more than 300 milliseconds without optimizations. The proposed optimization techniques improve performance by a factor of 3.49 on average, compared to an implementation without optimizations.

Observation: The performance with optimizations is quite similar for datasets of 1K and 4K, while performance degrades as the data size increases from 16K to 64K; this is especially noticeable for the deletion shown in figure 7(c). However, the performance without optimizations is very stable.

Explanation: While data size is less than 4K, most of the tree nodes in a tree can be cached, because some strings can share a common prefix. While data size is quadrupled to 16K and further to 64K, our cache starts to swap in and out tree nodes, incurring some overhead.

The aggregation can explain the performance degradation in the deletion shown in figure 7(c). As the data size becomes larger, it is more likely for multiple strings to share a common prefix. That means, during string deletion, there are

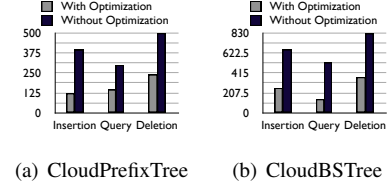


Figure 10. Subfigure (a) and (b) present performance of a CloudPrefixTree and a CloudBSTree respectively. We use 16 thousand data items in the tests. In tests with optimizations, we use a small cache of only 1000 cache lines, but with prefetching *disabled*. Time cost shown on vertical axis is in milliseconds.

more tree node updates (removal of a child from a node) operations than tree node deletion operations (a tree node contains only one child and removal of the child is equivalent to deleting the whole tree node). With optimizations, we aggregate multiple node deletions into one bulk delete, but we do not aggregate multiple update operations due to the lack of such mechanism in DynamoDB. Due to possible data consistency errors during parallel reads, we do not represent a node update operation with a combination of a deletion and an insertion. Therefore, aggregation for node deletions is more aggressive when deleting a string from a small dataset than from a bigger dataset. On the other hand, *without* optimizations, each tree node deletion or update incurs separate communication to the cloud.

Other Tests: Figure 9(a) and 9(b) present the performance of CloudPrefixTree with strings of length 16 and 32. We observe the consistent performance gains in these tests as we did with string length of 8. We also test the tree with a small cache of 1000 lines (with prefetching disabled) in figure 10(a), in which caching and aggregation are still effective while performance gains degrade due to the small cache size.

Analysis: In addition, we verify that CloudPrefixTree operations preserve the traditional complexity of $O(d)$, where d is the length of the string in the tree. Without optimizations, as we increase the string size from 8 to 16 and further to 32, we observe the time cost for tree operation is increased linearly, proportional to the number of communications between the client and the cloud. Under optimizations, as we increase the string size in the same way, the time cost for tree operation is increased sublinearly, primarily because the use of caching, prefetching, and aggregation reduces communications to the cloud.

C. CloudBSTree Performance

Figure 8 describes the performance of CloudBSTree. We observe that optimizations are effective in reducing communications between client and the cloud, thus improving performance. On average, a CloudBSTree with the optimizations is 5.5 times quicker than the implementation without the optimization. As shown in figure 8(c), in the test with

64K data items, the performance is improved by a factor of 6.5 by using optimizations.

Observation: The insertion and query operations cost roughly 70 milliseconds, while deletion takes 228 milliseconds for the 64K dataset.

Explanation: Queries, insertions, and deletions in a binary search tree logically have the same complexity of $O(\log(n))$. However, the deletion operation incurs more cloud communications. To delete an item D in a tree, we first locate the tree node Cur that contains D . If Cur is not a leaf, we have to find the largest item ML in Cur 's left subtree. Next, we replace D in the node Cur with the value ML . At last, we delete ML in the tree. All updates and deletions mentioned above require cloud communications.

In addition, we verify that the CloudBSTree preserves its traditional logarithmic nature. As we increase the data size from 8K to 64K in figure 8, we observe the time cost for tree operations is increased logarithmically. The total time cost is proportional to the number of cloud communications. Note that as the data size increases from 8K to 64K, the cache buffer starts to swap in and out tree nodes, which incurs overhead.

Other Tests: Figure 9(c) shows CloudBSTree performance using a *random* dataset of 32K unique integers. We also perform experiments with a *random* dataset of one Million unique integers. Under optimizations, query and insertion take 250 milliseconds on average and deletion takes 630 milliseconds on average with the dataset of one Million random integers. We observe consistent performance gains with these datasets. In addition, in figure 10(b), we present the performance with a small cache of 1000 cache lines and with prefetching disabled. The proposed optimizations continue to be effective to reduce communication cost.

VII. CONCLUSION AND FUTURE WORK

In this work, we propose a library *CloudTree* that enables users on a client computer to organize big data into tree data structures of their choice that are physically stored in the cloud. We use caching, prefetching and aggregation optimizations in the design and implementation of CloudTree to enhance performance. We have implemented the service of Binary Search Trees (BST) and Prefix Trees as current members in CloudTree library and have benchmarked their performance using the Amazon Cloud.

In the future, we continue to improve the performance of CloudTree by using compression, or different caching policies. For example, we could load the top n levels of a tree into the cache. We will also be implementing the cloud B-tree and R-tree members for the CloudTree library.

REFERENCES

- [1] A. Aggarwal and J. S. Vitter, "The I/O complexity of sorting and related problems (extended abstract)," in *Proceedings of the 14th International Colloquium on Automata, Languages and Programming (ICALP)*, 1987, pp. 467–478.
- [2] J. S. Vitter, *Algorithms and Data Structures for External Memory*, ser. Foundations and Trends in Theoretical Computer Science. Hanover, MA: Now Publishers, 2008.
- [3] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Proceedings of the 40th Annual Symposium on Foundations of Computer Science (FOCS)*, 1999, pp. 285–298.
- [4] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys (CSUR)*, vol. 39, no. 1, 2007.
- [5] H. Attiya and J. Welch, Eds., *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2004.
- [6] V. K. A. G. Ananth Grama, George Karypis, Ed., *Introduction to Parallel Computing*. Addison-Wesley, 2004.
- [7] A. Papadopoulos and D. Katsaros, "A-Tree: Distributed Indexing of Multidimensional Data for Cloud Computing Environments," in *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2011, pp. 407–414.
- [8] F. Dehne, Q. Kong, a. Rau-Chaplin, H. Zaboli, and R. Zhou, "A distributed tree data structure for real-time OLAP on cloud architectures," in *Proceedings of the IEEE International Conference on Big Data*, 2013, pp. 499–505.
- [9] Y. Yu, Y. Zhu, W. Ng, and J. Samsudin, "An Efficient Multidimension Metadata Index and Search System for Cloud Data," in *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2014, pp. 499–504.
- [10] S. Wu, D. Jiang, B. B. C. Ooi, and K. K.-L. Wu, "Efficient B-tree Based Indexing for Cloud Data Processing," *The Proceedings of the VLDB Endowment (PVLDB)*, vol. 3, no. 1-2, pp. 1207–1218, 2010.
- [11] C. R. Chang, M. J. Hsieh, J. J. Wu, P. Y. Wu, and P. Liu, "HSQL: A highly scalable cloud database for multi-user query processing," *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, pp. 943–944, 2012.
- [12] Z. Mo, Q. Xiao, Y. Zhou, and S. Chen, "On Deletion of Outsourced Data in Cloud Computing," in *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD)*, 2014, pp. 344–351.
- [13] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," Boeing Scientific Research Labs, Mathematical and Information Sciences Report 20, 1970.
- [14] P. Ferragina and R. Grossi, "The String B-tree: A new data structure for string search in external memory and its applications," *Journal of ACM*, vol. 46, no. 2, pp. 236–280, 1999.
- [15] M. A. Bender, M. Farach-Colton, and B. Kuszmaul, "Cache-oblivious string B-trees," in *Proceedings of the ACM conference on Principles of Database Systems (PODS)*, 2006, pp. 233–242.
- [16] A. Behm, C. Li, and M. J. Carey, "Answering approximate string queries on large data sets using external memory," in *Proceedings of the 2011 IEEE International Conference on Data Engineering (ICDE)*, 2011, pp. 888–899.
- [17] J. Baron and S. Kotecha, "Storage options in the aws cloud," *Amazon Web Services, Washington DC, Tech. Rep.*, 2013.
- [18] G. R. Mewhinney and M. S. Srinivas, "Optimized least recently used lookup cache," Sep. 2012, uS Patent 8,275,802.
- [19] Amazon, "Provisioned Throughput in Amazon DynamoDB," 2015. [Online]. Available: <http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ProvisionedThroughputIntro.html>